Homework #4: Substitution based Interpreter – Expanding the FLANG Language

Out: Tuesday, June 02, 2020, Due: Friday, June 12, 2020, 11:55 (before noontime)

Administrative

The language for this homework is:

```
#lang pl 04
```

The homework is basically about adding conditionals to the implementation of the FLANG interpreter– In the substitution model (use the interpreter we have seen in class as a basis for your code – see here).

Important: the grading process requires certain bound names to be present. These names need to be global definitions. The grading process *cannot* see names of locally defined functions.
This homework is for work and submission in pairs (individual submissions are also fine).

In case you choose to work in pairs:

1. **Make sure to specify the ID number of both partners within the file-name for your assignment. E.g., 022222222_44466767_3.rkt.**
2. **Submit the assignment only once.**
3. **Make sure to describe within your comments the role of each partner in each solution.**

Integrity: Please do not cheat. You may consult your friends regarding the solution for the assignment. However, you must do the actual programming and commenting on your own (as pairs, if you so choose)!! This includes roommates, marital couples, best friends, etc… I will be very strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Comments: Submitted code for each question should include at least two lines of comments with your personal description of the solution, the function and its type. In addition, you should comment on the process of solving this question – what were the main difficulties, how you solved them, how much time did you invest in solving it, did you need to consult others. **A solution without proper comments may be graded 0.** In general, comments should appear above the definition of each procedure (to keep the code readable).

If you choose to consult any other person or source of information, this **must be** clearly declared in your comments.

Tests: For each question, you should have enough test cases for complete coverage (DrRacket indicates covered expressions with colors for covered and uncovered source code, unless your code is completely covered). See below on the way to create tests. ***Important:*** Your tests should cover your whole code; otherwise the server will heavily penalize your submission. You should not have any uncovered expressions after you hit "Run" — it should stay at the same color, indicating complete coverage. Furthermore, the server will run its own tests over your code, which means that you will not be able to submit code that does not work. Reminder: this means that much of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests. Note that your tests should not only cover the code, but also all end- cases and possible pitfalls.
General note: Code quality will be graded. Write clean and tidy code. Consult the Style Guide, and if something is unclear, ask questions on the course forum.

The code for all the following questions should appear in a single .rkt file named <your IDs>_4 (e.g., 022222222_44466767_4.rkt for a pair of students with ID numbers 022222222 and 44466767, and 333333333_4 for a single-submission of a student whose ID number is 333333333).

1. Introduction

In class we saw the FLANG language and interpreter. It allows simple arithmetic expressions, it allows binding identifiers to expressions, and it provides a first class treatment for functions. It does not, however, allow for conditional expressions and Boolean values.

In the following, we will expand the FLANG (see here) language to also treat conditionals. Our goal is to allow logical operators and expressions to deal with Boolean (logical) values. In general, our treatment will have a very similar semantics to that of the pl language. Specifically, we will add **if** expressions, the binary (numeric to Boolean) operators <, >, =, the unary operator (Boolean to Boolean) **not**, and the Boolean values True and False (the expressions **#t** and **#f**

will not be part of our language). In addition, we will allow the run interface to return to the user non-numeric values.

Here are some tests that should work after you are done:

```
;; tests
(test (run "True") => true)
(test (run "{not True}") => false)
(test (run "{> 3 44}") => false)
(test (run "{if {- 3 3} {then-do 4} {else-do 5}}") => 4)
(test (run "{with {x 8}
                  {if {> x 0} {then-do {/ 2 x}} {else-do x}}}") => 1/4)
(test (run "{with {x 0}
                  {if {> x 0} {then-do {/ 2 x}} {else-do x}}}") => 0)
(test (run "{if {> 2 1} {then-do True} {else-do {+ 2 2}}}") => true)
(test (run "{with {c True}
                  {if c {then-do {> 2 1}} {else-do 2}}}")
      => true)
(test (run "{with {foo {fun {x}
                  {if {< x 2} {then-do x} {else-do {/ x 2}}}}} foo}")
      => (Fun 'x (If (Smaller (Id 'x) (Num 2)) (Id 'x) (Div (Id 'x) (Num 2)))))
(test (run "{with {x 0}
                  {if {> x 0} {/ 2 x} x}}")
      =error> "parse-sexpr: bad `if' syntax in (if (> x 0) (/ 2 x) x)")
 (test (run "true") =error> "eval: free identifier: true")
(test (run "{< false 5}") =error> "eval: free identifier: false")
(test (run "{< False 5}")
      =error> "Num->number: expected a number, got: #(struct:Bool #f)")
```

A remark about semantics of if expressions: Note that a conditional **if** expression cannot be replaced by a function. That is, you cannot just define a function named ‘if’ and apply it to three arguments. This is because the semantics of an **if** expression is “lazy” in the sense that it only evaluates the

“then-do” part if the condition is satisfied and the “else-do” part if the condition is not satisfied (but never evaluates both).

2. Expanding the FLANG BNF language

Extend your BNF and Parser to Support this syntax. Note that `**then-do**’ and `**else-do**’ are terminals and are part of the syntax of the “**if**” special form.

```
#|  The grammar:
    <FLANG> ::= <num>  ;; Rule 1
              | { + <FLANG> <FLANG> } ;; Rule 2
              | { - <FLANG> <FLANG> } ;; Rule 3
              | { * <FLANG> <FLANG> } ;; Rule 4
              | { / <FLANG> <FLANG> } ;; Rule 5
              | { with { <id> <FLANG> } <FLANG> } ;; Rule 6
              | <id> ;; Rule 7
              | { fun { <id> } <FLANG> } ;; Rule 8
              | { call <FLANG> <FLANG> } ;; Rule 9
              | —«fill-in 1»— ;; add rule for True  ;; Rule 10
              | —«fill-in 2»—                       ;; Rule 11
              | —«fill-in 3»— ;; add rule for =     ;; Rule 12
              | —«fill-in 4»—                       ;; Rule 13
              | —«fill-in 5»—                       ;; Rule 14
              | —«fill-in 6»—                       ;; Rule 15
              | —«fill-in 7»— ;; add rule 16 for (the above) if
expressions
|#
```

3. Extending the Parser

```
Use the above test examples to complete the missing parts of the FLANG
type definition and the parse-sexpr procedure.

 (define-type FLANG
    [Num  Number]

 ...   Original interpreter’s code omitted...

   [Call FLANG FLANG]
    [Bool  <--fill in 1 -->]
    [Bigger <--fill in 2 -->]
    [Smaller <--fill in 3 -->]
    [Equal <--fill in 4 -->]
    [Not <--fill in 5 -->]
    [If <--fill in 6 -->])

 (: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
```

```
    (match sexpr
      [(number: n)     (Num n)]
      ['True (Bool true)]
      ['False <--fill in 1-->)]
      [(symbol: name) (Id name)]

...   Original interpreter's code omitted...

      [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
      [(list '= lhs rhs) (Equal  <--fill in 2 -->)]
      [(list '> lhs rhs) <--fill in 3 -->]
      [<--fill in 4 -->]
      [(list 'not exp) <--fill in 5 -->]
      [(cons 'if  <--fill in 6 -->]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

4. Extending subst and eval

Use the following formal rules to complete the code for the `subst` procedure.

```
Formal Substitution rules:

    subst:
      N[v/x]                 = N
      {+ E1 E2}[v/x]         = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]         = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]         = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]         = {/ E1[v/x] E2[v/x]}
      y[v/x]                 = y
      x[v/x]                 = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
      {call E1 E2}[v/x]      = {call E1[v/x] E2[v/x]}
      {fun {y} E}[v/x]       = {fun {y} E[v/x]}            ; if y =/= x
      {fun {x} E}[v/x]       = {fun {x} E}

      B[v/x]                 = B      ;; B is Boolean
      {= E1 E2}[v/x]         = {= E1[v/x] E2[v/x]}
      {> E1 E2}[v/x]         = {> E1[v/x] E2[v/x]}
      {< E1 E2}[v/x]         = {< E1[v/x] E2[v/x]}
      { not E}[v/x]          = {not E[v/x]}
      {if Econd {then-do Edo} {else-do Eelse}}[v/x]
                            = {if Econd[v/x] {then-do Edo[v/x]} {else-do
Eelse[v/x]}}

 (: subst : FLANG Symbol FLANG -> FLANG)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (define (subst expr from to)
    (cases expr
...   Original interpreter's code omitted...
```

```
      [(Bool b) <--fill in 1 -->]
      [(Equal l r) <--fill in 2 -->]
      [<--fill in 3 -->]
      [<--fill in 4 -->]
      [<--fill in 5 -->]
      [<--fill in 6 -->]))
```

Use the following provided procedures and the formal rules below to complete the code below for the `logic-op` procedure and for the `flang->bool` procedure.

```
;; The following function is used in multiple places below,
   ;; hence, it is now a top-level definition
   (: Num->number : FLANG -> Number)
   ;; gets a FLANG -- presumably a Num variant -- and returns the
   ;; unwrapped number
    (define (Num->number e)
      (cases e
        [(Num n) n]
        [else (error 'Num->number "expected a number, got: ~s" e)]))

  (: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
  ;; gets a Racket numeric binary operator, and uses it within a FLANG
  ;; `Num' wrapper
  (define (arith-op op expr1 expr2)
    (Num (op (Num->number expr1) (Num->number expr2))))

  (: logic-op : (Number Number -> Boolean) FLANG FLANG -> FLANG)
  ;; gets a Racket Boolean binary operator (on numbers), and applies it
  ;; to two `Num' wrapped FLANGs
  (define (logic-op op expr1 expr2)
    <--fill in 1 -->)

  (: flang->bool : FLANG -> Boolean)
  ;; gets a Flang E (of any kind) and returns a its appropiate
  ;; Boolean value -- which is true if and only if E does not
  ;; represent false
  ;; Remark: the `flang->bool` function will also be top-level
  ;; since it's used in more than one place.
  (define (flang->bool e)
    (cases e
      [<--fill in 2 -->]
      [else <--fill in 3 -->]))
```

Use the above defined procedures and the formal rules below to complete the code below for the `eval` procedure. Consult the provided tests at the introduction part of the assignment.

```
   eval:  Evaluation rules:
      eval(N)             = N ;; N is an expression for a numeric value
      eval({+ E1 E2})     = eval(E1) + eval(E2)   \ if both E1 and E2
```

```
eval({- E1 E2})     = eval(E1) - eval(E2)    \ evaluate to numbers
eval({* E1 E2})     = eval(E1) * eval(E2)    / otherwise error!
eval({/ E1 E2})     = eval(E1) / eval(E2)   /
eval(id)            = error!
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(FUN)           = FUN ; assuming FUN is a function expression
eval({call E1 E2}) = eval(Ef[eval(E2)/x])
                        if eval(E1)={fun {x} Ef}
                    = error!                otherwise

eval(B)              = B ;; B is an expression for a Boolean value
eval({= E1 E2})        = eval(E1) = eval(E2)  \ if both E1 and E2
eval({> E1 E2})        = eval(E1) > eval(E2)   \ evaluate to
                                                      numbers
eval({< E1 E2})        = eval(E1) < eval(E2)   / otherwise error!
eval({not E})          = not(eval(E))         /E may be anything
eval({if Econd {then-do Edo} {else-do Eelse}})
                       = eval(Edo) if eval(Econd) =/= false,
                                   eval(Eelse),    otherwise.
```

Remark: The semantics of the not operation is defined by the `not' operation of pl.

```
 (: eval : FLANG -> FLANG)
  ;; evaluates FLANG expressions by reducing them to *expressions*
  (define (eval expr)
    (cases expr
...   Original interpreter's code omitted...

      [(Bool b) <--fill in 1 -->]
      [<--fill in 2 -->]
      [<--fill in 3 -->]
      [<--fill in 4 -->]
      [(If l m r)
       (let ([<--fill in 5 -->])
         (<--fill in 6 -->))]
      [(Not exp) [<--fill in 7 -->]]))
```

5. Extending the run procedure

Finally, we will allow the interface procedure to return any one of the three possible types of the extended language. Use the above test examples to complete the code for the `run` procedure.

```
(: run : String -> (U Number Boolean FLANG))
  ;; evaluate a FLANG program contained in a string
  (define (run str)
    (let ([result (eval (parse str))])
```

```
(cases result
  [<--fill in 1 -->]
  [<--fill in 2 -->]
  [<--fill in 3 -->]))))
```